SHARP APL *for MVS*

Guide for APL Programmers

Version 22

SOLITON ASSOCIATES

SHARP APL is a registered trademark of Soliton Associates Limited.
SVP, NSVP, and Mailbox are trademarks of Soliton Associates Limited.
All other products referred to in this document are trademarks or registered trademarks of their respective companies.

© 1997 by Soliton Associates Limited — Printed in Canada
44 Victoria Street, Suite 2100, Toronto, Ontario, Canada M5C 1Y2

All rights reserved. No part of this work may be reproduced in any form or by any means without prior written permission of the copyright owner.

Soliton Associates Limited shall not be liable for errors contained herein or for incidental consequential damages in connection with the use of this material. The information contained within this document is subject to change without notice.

Publication code: MM-000-1502 (9703)

Guide for APL Programmers

Introduction

SHARP APL Version 22 for MVS ("V22") provides a number of enhancements to the SHARP APL product, resulting in improved performance, increased availability, and enhanced operability of the SHARP APL system. These changes include modifications and additions to the base APL system and to the file system, as well as changes to SHARP APL utilities and other related software products.

This guide documents the changes that are of interest to APL programmers.

- SHARP APL year 2000 compliance as of Version 22.
- New features:
  - shared system variable □*to* used to set time-outs for various system suspensions
  - system function □*bind* used to create IFN binds
  - workspace `1 translit` available for APL character transliteration.
- Other features:
  - enhanced □*ai* returns *floating point* instead of *integer* (fixed-point) results
  - extended definition of APL packages includes *bound intrinsic functions*
  - improved *less* (~) function
  - improved Batch Task Scheduler (BTS) auxiliary processor.

Note: This manual is designed to be used with the *SHARP APL Reference Manual* and the *Guide for APL Programmers* for Version 21; these books provide the complete description of SHARP APL for MVS as it relates to programmers.

Year 2000

Soliton Associates has used good faith efforts to review SHARP APL for MVS. Necessary modifications have been made to system and utility software, file naming conventions, and data storage so that date calculations and date information are accurate across the century change.

SHARP APL Changes

This section details changes made to the structure of data sets and files used by the billing system, and to the SHARP APL utilities in preparation for the year 2000.

There have been several changes to the structure of data sets and files used by the billing system, and to the SHARP APL utilities to prepare for the year 2000 and beyond. Where two digit years continue to exist, if the year is in the range 50 to 99, it assumes years 1950 to 1999; if the year is in the range 0 to 49, it assumes 2000 to 2049. Changes to the various system components will enforce four digit years in future releases, where possible.

APL-Coded Software

The APL-coded software included in Version 22 has been updated to allow storage and representation of dates necessary to correctly deal with the century change. The changes include modification to the internal representation of the Usage data file (`314159 ohc`) and the formatted version of the file (`314158 ohcf`), as well as changes to a number of APL functions used in the Usage Inquiry system and elsewhere.

Note: Authors and users of private copies of APL-coded software supplied by Soliton are urged to read this section carefully, and consider the implications; because the underlying structure of the sign-off history files (offhist) has changed, and because many reporting functions in the Usage Inquiry System now behave differently, private copies may fail or provide incorrect results.

Changes to Files:

- `314159 ohcmmyy` (the unformatted offhist file) is named `314159 ohcmmyr` where *yr*=`1900` | *yyyy*; e.g., 1997 is `ohcmm97` and 2000 is `ohcmm100`.
- `314159 ohcfmmyy` (the formatted offhist file) is now named `314159 ohcfmmyr` where *yr*=`1900` | *yyyy*. ***New format directory entries*** are stored as *yyyymmddhhmm*, while ***old format directory entries*** remain *ymmddhhmm*. ***New format data entries*** store the date as *yyyymmdd*, while ***old format data entries*** remain *mmddyy*.
- `444 bfmmmyy` (the monthly consolidated usage/billing file) is now named `444 bfmmmyr` where *yr*=`1900` | *yyyy*. All status dates in component 1 are *yyyymmdd*.
- `1708363 bfind` (tracks daily and monthly billing runs) now stores the component 2 date (date of the last daily billing run) as *yrmmdd* where *yr*=`1900` | *yyyy*.

- 444 *numbreq* (*numbreq* data file) holds dates in component 17 as *yyyymmdd*. Directory components [;6+□*io*] for ***new format directory entries*** are *yyyymmdd*, while ***old format directory entries*** remain *yymmdd* and are assumed to be for years 19*xx*.

Summary of Billing File Name Changes

The files 314159 *ohc*, *ohcf* and *bfmmm* form the year portion of their names from the pattern: 1900 | *yyyy*, resulting in file names like *ohcf*100 and *bfmmm*100 for the year 2000. For example:

Year	*File Names*	*Year*	*File Names*	*Year*	*File Names*
1997	*ohcmm*97 *ohcfmm*97 *bfmmm*97	2000	*ohcmm*100 *ohcfmm*100 *bfmmm*100	2001	*ohcmm*101 *ohcfmm*101 *bfmmm*101

Changes to Functions and Variables

All software was modified, where appropriate, to recognize four-digit years, while compatibility was maintained with the two-digit format formerly used. The following table shows a summary of the changed functions and variables grouped by the workspace or file that contains them.

Workspace/File	*Function/Variable Name*	*Change*
1 *usage*	*ohprint*	*RA* can be *mm,yy* or *mm,yyyy*. If year is 2 digits then the 0-49 etc. rule is applied.†
	print	"
	time	Handles formatting of *mmddyy* and *yyyymmdd* dates.
1 *ts*	*ttf*	*RA* can have 2 or 4 digit year. If 2 digit year then 1900 assumed.
	lts	"
	utc	"
	getoffset	*LA* can have a 2 or 4 digit year. If year is 2 digits then 1900 is assumed.
	ttfhow(V)	modified to reflect change to function.
	ltshow(V)	"
	utchow(V)	"
	getoffsethow(V)	"

Notes:
RA=right argument.
LA=left argument.
(V)=variable.
† The 0-49, 50-99 Rule: if the year is in the range 50-99, it indicates years 1950 to 1999, if in the range 0-49, it indicates 2000 to 2049. (If yyyy is used, then yyyy is a simple year value, such as 1997 or 2000.)

New Features

□to Time-Out

The new shared system variable □to is used to set time-out limits for two kinds of system suspensions. At present, it consists of an $n \times 2$ numeric matrix. The first row controls time-outs for shared variable waits, and the second row controls time-outs for file system waits. Each row consists of two elements:

- an *event number*, specifying to which suspension the time-out applies, and which event is signalled if the timer expires.
- a *time-out value*, indicating the duration of the time-out in seconds. If the time-out value is zero, no time-out is imposed.

Event numbers are currently restricted to the values 1201 (SVP waits) and 1202 (file waits). Time-out values must be non-negative and have no imaginary part. If □to is set to an illegal value, the system responds by setting □to to the default, 2 2 □ρ1201 0 1202 0 (no time-outs imposed).

If a non-zero time-out is specified for a given type of suspension, and the suspension does not terminate before the time-out expires, the event number (first element of the applicable row of □to) is signalled. These events are *sv timeout* and *file timeout* respectively. They can be trapped via □trap, individually, or along with all other interrupts by trapping event 1000.

Examples:

```
      )clear
clear ws
      □to          ⍝ default value
1201 0
1202 0
```

The time-out value must be a non-negative integer representing the number of seconds to wait.

```
      □to[0;1]←10  ⍝ set shared variable timeout to 10 seconds
      □to
1201 10
1202 0
      □to←'abc'    ⍝ illegal set
      □to          ⍝ forces default value
1201 0
1202 0
      x←4 2ρ1201 1 1202 2 1201 3 1202 4 ⍝ multiple value for each
      x
1201 1
1202 2
1201 3
1202 4
      □to←x  ⍝ only last value for each is meaningful
      □to
1201 3
1202 4
```

Using the above for □to, a shared variable time-out might look like the following.

```
      123456789 □svo 'y'  ⍝ share with non-existent user
1
      1 □svc 'y'  ⍝ fully interlock
1 1 1 1
```

```
 y  ⍝ enter shared variable wait with 3 second timeout
<three seconds pass>
sv timeout interrupt
      ∇r←foo;⎕to
[1] r←⎕to
      ∇
      foo ⍝ value of ⎕to after localisation is default
1201 0
1202 0
      ⎕to  ⍝ global value restored upon function exit
1201 3
1202 4
```

If ⎕to is not explicitly set during a session, any reference to ⎕to will cause the system to attempt to create the variable ⎕to with its default value. If WSFULL is encountered, trying to create ⎕to for any reason (e.g., explicit set, localization, or any other attempt to set default values when ⎕to is undefined), ⎕to is left undefined, time-out delays are cleared, and WSFULL is signalled. As a result, WSFULL is the only error that can occur from a reference to ⎕to.

Notes on Using ⎕to and ⎕sc Together

This section explains the results of using the State Change variable, ⎕sc, as a timer for shared variable operations along with ⎕to.

When entering a ⎕sc wait, the system checks the ⎕sc delay value (if any) versus the ⎕to delay value (if any) and sets the timer to the shorter of the two. Upon return to ⎕sc from the delay, the actual time-of-day is checked versus the time-of-day at entry to the wait plus the delay set (if any). If the current time-of-day exceeds the previous time-of-day plus the delay, it is assumed that the timer expired (whichever timer was set).

If this is the normal ⎕sc timer, then you will see the normal result of referencing ⎕sc. If it is the ⎕to timer, event `1201` is signalled, even though it is possible that the wait was terminated by a shared variable post and the time required to schedule the task made it seem as though the ⎕to timer expired. Conversely, the timer may expire and cause event `1201` to be signalled, even though a shared variable post occurs before the event is actually signalled.

So, as regards ⎕sc, event `1201` means that the ⎕to timer expired before the ⎕sc wait terminated. Any number of ⎕sc posts may actually have occurred in this time, but the system was unable to terminate the ⎕sc wait before the timer expired.

⎕bind Create an IF Bind

The system function ⎕bind is used to create an association between an APL symbol/name and an intrinsic function, or to query the association level. This association, *intrinsic function name bind* or IFN bind, can also be achieved via `system.bind`. There is an association to `system.bind` called `IFbind` located in workspace `1 if`.

An intrinsic function is a function produced outside the APL workspace that can be associated with (*bound* to) a name in the APL workspace; that name can be called like any other APL function. Intrinsic functions are most commonly used to improve performance of SHARP APL applications.

A library of public intrinsic functions was supplied with SHARP APL Version 21. For complete information on the IF facility, please refer to the documentation provided with workspace `1 if`. Contact your Soliton Associates representative if you have any questions regarding the use or implementation of intrinsic functions.

Transliteration Utility

Distributed with this version of SHARP APL is a public workspace containing some standard APL functions that may be used for sending and receiving APL code over the Internet.

`1 translit` enables conversion of APL characters (glyphs) into ASCII-coded keywords and vice versa. This workspace, which implements Jim Weigang's industry-accepted APL-to-ASCII transliteration, is offered without charge and may be freely reproduced and/or used. All functions contained in this workspace are Copyright ©1996 by the author of the function.

Loading the workspace `1 translit`, then typing `describe` provides a complete description of the workspace and its documentation.

Note: This transliteration utility, `1 translit`, is not a SHARP APL product; it is distributed "as-is", and it is not supported by Soliton Associates Limited.

Other Features

□ai Accounting Information

The niladic system function □ai, which provides information about resource usage, now returns floating point results instead of

- signalling a domain error when fixed point overflow occurs, or
- (erroneously) producing results that contain negative elements.

Warning: Due to the enhancement of □ai, the results of some expressions may change.

Whenever any of the elements of □ai have grown too large to be represented in the *integer* data type, □ai returns a *floating-point* result. This has the potential to cause problems if the *format* primitive, ⍕, is used to display elements of □ai (refer to the section " Documentation Changes" for more information on the *format* primative.

Arrays that are of the internal *integer* data type are never formatted in scientific notation (e notation), regardless of the value of □pp. Thus, in the past, it was safe to use `⍕''⍴□ai` to construct a string for signing onto an S-task; for example, `')',⍕''⍴□ai`. This would produce the desired result since the result of □ai was guaranteed to be of the *integer* data type.

This is no longer the case. The result of the expression `')',⍕''⍴□ai` is now sensitive to the value of □pp. If we assume that the first element of the result of □ai is `1234567` and □pp is `6`, then, depending on the result type,

```
')',⍕''⍴⎕ai      ⍝ returns the following integer result.
)1234567
```

or, after account usage causes the result of ⎕ai to contain values too large to be represented as integers,

```
')',⍕''⍴⎕ai      ⍝ returns the following floating point result.
)1.23457e6
```

Obviously, the second result is not suitable as a sign-on string. This problem can be avoided by using floor (⌊) to ensure that the argument to display is in the *integer* datatype, i.e.,`')',⍕⌊''⍴⎕ai`; the first element of the result of ⎕ai is always representable in the *integer* data type, even if the result is in the *floating point* data type. Note that is not the case for any other element of the result of ⎕ai.

APL Packages

The definition of an APL package has been extended to include *intrinsic function binds*, as well as variables and user-defined functions. ⎕pnc for a bind is ¯2. A bind is treated much like a locked function. NSVP/MVS 1.3 does not support transmission of a package containing IF binds

See chapter 4 of the *SHARP APL Reference Manual* for further information on APL packages. System functions for manipulating packages, ⎕pack, ⎕pdef, ⎕pex, ⎕pins, ⎕plock, ⎕pnames, ⎕pnc, ⎕ppdef, ⎕psel, and ⎕pval, are detailed in chapter 16 of the *SHARP APL Reference Manual.*

Less (~)

Dyadic tilde (~), defined as the primitive function *less,* is enhanced to properly handle edge conditions, and to correct errors reported for the previous version of this function (Version 21).

Batch Task Scheduler

The Batch Task Scheduler, BTS, which is responsible for scheduling, dispatching, and restarting B-tasks and scheduled N-tasks, now provides a 10-second delay between the time it detects that it will shut down and the time it actually starts to do so. This enhancement allows B-tasks enough time to terminate on their own, rather than as the consequence of a BTS shutdown.

The configuration for setting a delay in the Batch Task Scheduler is documented in Chapter 15 of the *SHARP APL for MVS Auxiliary Processor's* Manual.

Documentation Changes

SHARP APL Reference Manual

The following items are outstanding corrections or additions to the *SHARP APL Reference Manual*, 2nd edition. Where appropriate, page references are provided.

Under (⍣⍣) Requires a Known Inverse

On page 7–16, the list of primitive monads and their inverses that can be used as a right argument to *under* (⍣⍣) should read as follows:

Monad:	⋆	⊛	>	<	−	÷	⍟	⌹	~	⊢
Inverse:	⊛	⋆	<	>	−	÷	⍟	⌹	~	⊢

`(2 ⎕ws 3)[4]`

On page 16–87, the system function `(2 ⎕ws3)[4]` is incorrectly documented as being unused. The correct meaning of this element is as follows:

`(2 ⎕ws 3)[4] 1` if hot continue was loaded at sign-on; remains `1` until sign-off.
`0` if no hot continue was loaded.

`6I12`

Page 16–90 incorrectly documents the availability of `6I12 1` and `6I12 3`. These are in fact not supported. The only two variations of `6I12` that are supported are described below.

`6I12 ¯1` Same effect as `)keyb lock`.

`6I12 0` Same effect as `)keyb nomsg`.

Format Primitive (⍕)

New text should be added to page 6–55 of the *SHARP APL Reference Manual*.

The section "*Choice of Fixed-Point or Exponential Format for Real Numbers*," includes a new bullet after "*When ⍵ is real, SHARP APL selects a format as follows:*".

- A number whose internal representation is integer is represented in *fixed-point* format.

Reload User Profile:)proj

The following new text should be added to the *SHARP APL Reference Manual.*

The `)proj` command automatically loads an account's `Profile` workspace. This workspace is normally loaded when the task signs on, if the user profile list has been set for this account. However, the `Profile` workspace would not be loaded if a "hot continue" workspace is loaded instead.

Performing `)proj` is equivalent to performing `)load Profile`.

Version 21 Guide for APL Programmers

The following item is a correction to the *Version 21 Guide for APL Programmers.*

File Copy Function □copy

Page 3 lists incorrect numbers for the ***File*** and ***Library*** permissions (the numbers are reversed). The correct text is as follows:

Library permission: `64`
File permission: `131072`